Towards Activity-Aware Tool Support for Change Tasks

Kevic, Katja ; Fritz, Thomas

DOI: https://doi.org/10.1109/ICSME.2017.48

Towards Activity-Aware Tool Support for Change Tasks

Katja Kevic, Thomas Fritz
Department of Informatics, University of Zurich, Switzerland
{kevic, fritz}@ifi.uzh.ch

***Abstract*—To complete a change task, software developers perform a number of activities, such as locating and editing the relevant code. While there is a variety of approaches to support developers for change tasks, these approaches mainly focus on a single activity each. Given the wide variety of activities during a change task, a developer has to keep track of and switch between the different approaches. By knowing more about a developer's activities and in particular by knowing when she is working on which activity, we would be able to provide better and more tailored tool support, thereby reducing developer effort.**

In our research we investigate the characteristics of these activities, whether they can be identified, and whether we can use this additional information to improve developer support for change tasks. We conducted two exploratory studies with a total of 21 software developers collecting data on activities in the lab and field. An empirical analysis of the data shows, amongst other results, that activities comprise a consistently small amount of code elements across all developers and tasks (approx. 8.7 elements). Further analysis of the data shows, that we can automatically detect the boundaries and types of activities, and that the information on activity types can be used to improve the identification of relevant code elements.

I. Introduction

Software developers spend a substantial amount of their time working on change tasks, such as bug fixes or enhancements [1]. We use the terms *change task* and *task* to refer to any change to a software system that serves a predefined purpose, such as a bug fix or an enhancement. During these change tasks, developers perform a variety of steps and activities as well as address multiple questions as previous research studies have shown [2]–[4]. For instance, to complete a bug fixing task, a developer needs to perform a number of distinct activities. First she might start out by locating the bug in the code base using search and navigation. Second, she might thoroughly examine the involved code elements and their relations as well as investigate related documentation to come up with a way to fix the bug, before she finally edits the code and commits the changes to the repository. Each of these activities requires different kinds and granularity of information, ranging from a list of code search results, to individual method calls, all the way to API documentation or stackoverflow posts for making code changes.

To support developers in their work on change tasks, various approaches have been proposed to help identify relevant information, such as code search tools for the initial search, recommenders for the code navigation or code completion tools for the editing of code. Predominantly, each of these approaches focuses on supporting one specific activity during the work on a change task to determine relevant information without adapting to the various activities the developer is performing over the course of the change task. With the wide variety of developer activities during a change task [2], [4] and change tasks lasting anywhere from a few minutes to several days [5], finding the right approach and the relevant information as well as switching to it is difficult at best [6], [7]. The more we know about developers' activities during change tasks, the better we can support them in their work. For instance, if we know that a developer is looking for an initial focus point rather than understanding the behavior or trying to edit a specific code element, we might be able to recommend broader code search results, rather than providing more fine-grained information on code dependencies or code snippets from stackoverflow for the correct editing.

To better support developers in their work on a change task and help identify relevant information at the right time, we investigate three questions:

RQ1: What are the characteristics and types of developers' activities on change tasks? Previous research has inferred activities developers perform on various levels of granularity [2], [3], [7], [8] as well as the questions that developers ask during change tasks [4]. These inferred activities or questions range from low-level developer actions, such as using a navigation tool to high-level questions, such as the dependency between modules. The goal is to extend previous studies by exploring the *set of basic activities that developers' themselves break their work on a change task into* as well as the characteristics of these small units of work, in particular their size, granularity and possible types. The better we understand the units developers work and think in—what we will refer to as "activities" in the following—the better we can support them in their work and make recommendations on relevant information or tools.

RQ2: How accurately can we automatically detect (a) the boundaries of developers' activities and (b) the types of these activities during a change task? By knowing more about the activity that a developer is working on rather than just the high-level change task, we might be able to tailor recommendations to the specific activity and thereby provide more relevant recommendations while also requiring less effort from the developer. The goal of RQ2 is to explore how accurately we can detect the activity a developer is working on and when the developer is switching to another activity during the work on a change task.

RQ3: Can we use activity information to more accurately identify relevant code elements for a change task? The goal of RQ3 is to explore the value of knowing about the

activities a developer is working on during a change task. For this, we are focusing on the identification of relevant code elements—a scenario that is commonly addressed in research to support developers during change tasks [9]–[11]—and examine whether we can enhance previous approaches by taking advantage of the additional knowledge on the activities that developers are working on.

To address these questions, we conducted two exploratory studies: a field study with nine professional developers working on their own change tasks, and a lab study with twelve developers working on two open source change tasks. For both studies, we collected developers' self-reports on the activities they broke their change tasks into. In addition, we collected their low-level code interactions within their Integrated Development Environment (IDE), such as selections and edits of methods and classes. Based on an empirical analysis of the collected data, we found that there is a small set of basic activity types across all participants that is similar to the ones identified in previous research [4], [12], including *understanding a specific code element* and *understanding a larger context*. The self-reported activities encompass on average 9 code elements (classes and methods) that were explored and a bit more than a third of these are relevant for the activity (RQ1). By applying regression analyses, we found that it is possible to automatically detect activity types and boundaries—when a developer starts/ends to work on an activity—with more than 75% accuracy in both cases. (RQ2). Finally, a comparative analysis shows that the use of activity information can improve precision and recall for recommending relevant code elements by 33% and 57% respectively (RQ3).

This paper makes the following contributions:

- it examines the types and characteristics of developers' self-reported activities for change tasks based on two exploratory studies with 21 developers and related work;
- it demonstrates that an activity's boundaries and type can automatically be detected with high accuracy;
- it illustrates the potential of using activity information to better support developers in their work on change tasks.

II. Related Work

Work related to our research can be grouped into empirical studies on developers' activities during change tasks, approaches to identify tasks, and approaches to identify the relevant source code elements for a change task.

A. Studies on Developers' Activities During Change Tasks

Several empirical studies have been conducted to better understand developers' work on a change task. These studies vary in the granularity of the developer activities they focus on, ranging from very abstract and high-level activities, such as understanding and editing code [2], [3], [7], [8], [12]–[15], and the high-level questions developers ask ("To move this feature into this code what else needs to be moved?" [4]), all the way to very low-level activities, such as microtasks [16], activities for incremental change [17] or interactions with code elements (classes, methods and even lines) and commands used [7], [8], [18]–[22]. Soh et al. [23] examine developers' efforts on such activities. Based on an extensive literature survey of empirical studies in the area, we came up with a coding of developer activities that is illustrated in Figure 1. Note that this figure presents one way of classifying developer activities that emerged from our coding of previous study findings. It does not capture all activities identified in these studies, also since different studies often used the same terminology for various activities, e.g. navigating and searching [7], [8] or different terminology for the same activity, e.g. "What is the mapping between these UI types and these model types?" [4] and gain high level overview of program [12].

These empirical studies on developers' activities can also be categorized by the methods used in the studies ranging from fine-grained logging to observations and surveys. Several studies instrumented the IDE to capture and log developers' low-level interactions with code elements and then used these logs to infer developers' activities and time spend on them [7], [8], [21]. To capture more of the developers' thought process, several studies used a think-aloud protocol in combination with observations and in-person shadowing or audio recordings to identify the questions developers' ask, and their information needs [2], [4], [14]. Finally, researchers have also used surveys to elicit developers' work practices [3], [14].

In our research, we use a combination of developers' self-reports and low-level interaction logging to explore the activities that developers themselves break their tasks into rather then inferring them retrospectively. Further, we examine the automatic detection and use of these activities.

B. Task Detection

Researchers have also investigated the manual [24] and automatic [25] detection of whole change tasks. Closest to our work is the approach by Coman and Sillitti [26] that looked at detecting tasks within a recorded navigation sequence. While they evaluated their proposed algorithm in a laboratory setting, Zou and Godfrey [27] reproduced their work in an industrial setting and found that there are levels of activities below a change task for which the detection might be more accurate. Instead of analyzing developer interactions, Barnett et al. [28] examined source code element definitions and usages within change sets to predict when a change set includes unrelated code changes and does not belong to the task.

In our work, we focus on a lower level, namely the activities into which developers break their change tasks into and on how accurately we can detect these automatically.

C. Relevancy Assessment of Code Elements

Developers spend a substantial amount of their time searching, navigating, and reading source code to locate and keep track of the code elements that are relevant to their work [2], [7], [8]. Several approaches emerged to support developers in identifying these relevant code elements more efficiently, for instance, during task resumption [9] or code navigation [29]. What all of these approaches have in common is a relevancy model that captures the relevancy of individual code elements,

yet, the way the relevancy is calculated differs between approaches. Researchers have proposed to use various data sources for creating the relevancy models, ranging from program structure [30]–[32], lexical similarities of code elements and change task descriptions [33], frequency and recency or 'momentum' of developers' code interactions [9], [34], version histories [35], [36], or combinations of some of these sources [37]. Similar to Mylyn [9], we use the interactions of developers' with the IDE to determine the relevancy of code elements. While Mylyn focuses on change tasks, we complement it by investigating the lower level activities that developers perform during change tasks, analyzing how the information on these activities can be used to improve the relevancy recommendation, and by automatically detecting switches between these lower level activities rather than having the developer explicitly indicate the start and end of a task. In a study comparing multiple of these sources, Piorkowski et al. [38] found that approaches recommending recently and frequently visited elements performed best.

Different to previous work, we investigate the use of a new kind of information—the information on developer activities—and how we can use this to complement more traditional sources in the identification of relevant code elements.

III. Study Method

To investigate how developers decompose change tasks into activities and the automatic detection of these, we conducted a lab and a field study with a total of 21 software developers. The lab study allowed to control for the change tasks and to examine how different developers decompose the same change tasks. In the field study, we examined how professional developers decompose their regular change tasks in a real work environment. In both studies we logged participants' source code interactions in the IDE and gathered data on their activities either through self-reporting or periodic interruptions.

A. Lab Study

Change Tasks. All participants in this study worked on the same two real change tasks of the open source system Gson [39]. We selected the two change tasks—the feature request with id #42 and the defect with id #153—based on them being already resolved, yet from an actively maintained project, having a commit history as well as their reasonable scope of the solution and the effort required to reproduce/test the task. The change set for feature request #42 included changes to several classes, while the changes to fix defect #153 were located in multiple methods of a single class (see Table I). Gson is a Java project that (re)converts Java objects into JSON strings, has a total of 31.7K lines of Java code not including comments and is composed of 159 Java files.

Participants. Through personal contacts, we recruited twelve participants (one female, eleven male) from our institution: one postdoctoral researcher, nine graduate and two undergraduate students. All twelve participants had their major in computer science, on average 8.1 ($\pm$5.1) years programming and 3.9 ($\pm$3.9) years professional programming experience, and were familiar with the Eclipse IDE (eight also stated that Eclipse is the IDE they are most familiar with).

Procedure. The study lasted on average 90 minutes and had a preparation, a training, and a programming phase. In the training phase participants worked on a change task to get familiar with the Gson project. In the programming phase participants worked on a different change task and we gathered detailed data on their activities. We had two groups that we randomly assigned participants to, one group of six started with task #42 (training) and then worked on task #153 (programming), the other group worked on the tasks in reverse order. We changed the task order to counteract any specific learning effect and to capture activities related to different kinds of change tasks. We prepared an Eclipse IDE instance in which we imported the Gson project into and installed our monitoring plugin.

In the preparation phase, we asked participants to read and sign a consent form and to complete a questionnaire on demographics. Then, we explained our goal of studying developers' work breakdown for a change task, the study procedure, answered questions regarding the procedure, provided a Gson overview, and allowed participants to explore the subject system for a few minutes. In the training phase, we asked participants to work on the first task for 20 minutes to get more familiar with the code.

In the programming phase, we asked participants to work on the second change task for 25 minutes. We asked participants to think-aloud and to report in our plugin "whenever they started working on something new". We intentionally did not specify the phrase "work on something new" any further since we were interested in capturing the units that developers work and think in. In case participants returned to work on something they had previously reported, they could just select the previously reported activity in the plugin to ease the reporting. We added the think-aloud protocol after we noticed in our pilot study that participants forgot to report some activities without it. After 25 minutes, we asked participants to go over all reported activities. Using our plugin, we presented participants a time-ordered list of the code elements they interacted with, and asked them to identify the ones that were relevant for each activity.

Plugin. We developed a plugin to log all user interactions with code elements (classes and methods) in an IDE, similar to Mylyn's interaction monitor [9]. Our plugin also has a user interface to report activities during a change task (Figure 2), and one to select the code elements relevant for an activity.

B. Field Study

Change Tasks. All participants in this study worked on their usual change tasks at work. Participants worked on a variety of tasks, such as fixing bugs, adding new features, and writing unit tests. The participants reported that they spend on average 10.5 ($\pm$3.4) hours to complete a change task. The projects participants worked on comprised on average 244K ($\pm$572K) lines of Java code. For privacy reasons, we cannot disclose any further information on the tasks or projects.

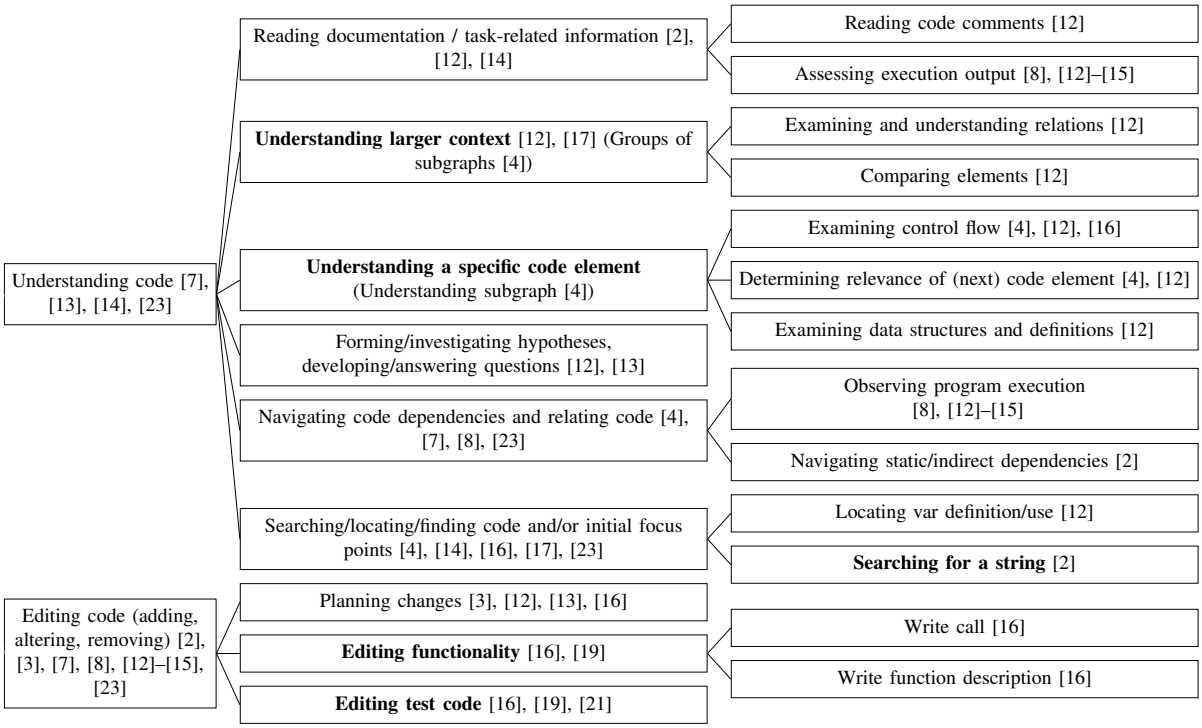


Fig. 1: Overview of developer activities within the IDE based on a coding of related work. (Elements in bold are the activity types we identified, see section IV)

TABLE I: Tasks used in the lab study.

ID	Date Submitted	Title	Scope of solution committed to the repository
42	9/8/2008	provide a feature to protect against remote "script src" inclusion of Gson output	multiple classes in which multiple methods were affected: `Gson`, `GsonBuilder`, `JsonParserImpl`, `JsonParserImplConstants`, `JsonParserImplTokenManager`
153	9/2/2009	setPrettyPrinting cause missing comma delimiter after an empty map	13 methods in a single class: `JsonPrintFormatter`

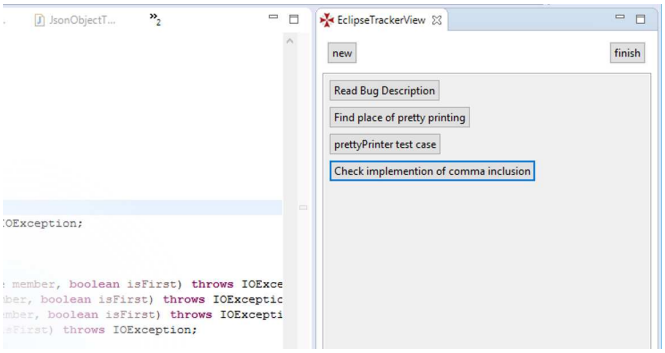


Fig. 2: Screenshot of our plugin used in the programming phase of the lab study to report and visualize activities.

Participants. We used recruiting emails and ended up with nine professional developers (one female, eight male) from two medium-sized software development companies and a total of four different development sites where they were either working in an open-plan or in private offices. Participants had an average of 8.6 ($\pm$6.0) years of professional programming experience and all nine were using Eclipse for their daily work.

Procedure. The field study had a preparation phase and two working sessions of two hours each, capturing a total of four hours of the work of participants. The working sessions were spread over two different days. In the preparation phase, we again asked participants to read and sign a consent form and to complete a questionnaire on demographics. We further explained that we wish to study how developers decompose change tasks and how developer tools might profit from this knowledge. We then explained the study procedure and gave them the opportunity to ask any questions they had. In the preparation phase, we also installed or asked participants to install our plugin into their IDE. Similar to the lab study, our plugin logged interactions with source code elements. It also provided features to visualize a time-ordered list of the code elements they interacted with such that they could identify the starting point of the current activity, and allowed to mark the code elements that were relevant for the current activity.

For the two working sessions of two hours each, we picked times when participants had no meetings scheduled, but otherwise asked participants to work as per usual. Most of the participants got at least once interrupted during the study sessions by a colleague, an email which caught their attention or an instant message which they quickly answered. After 30 minutes, we interrupted participants and asked them to:

(a) write down *what they were just working on*,
(b) identify the source code interaction when they started working on the just reported activity using a time-ordered list of their interactions, and
(c) identify the relevant code elements for this activity from the list of source code interactions.

As in the lab study, we did not provide any more detailed specification or examples of what to write down, since we were interested to study and capture how developers themselves decompose the work for a change task. We repeated this procedure a second time in each session and ended up with a total of four interruptions over the two sessions. To minimize the time and impact of our study on their usual work and avoid disrupting them or their colleagues any further, we chose not to employ a think-aloud protocol in the field study.

C. Data Collection

In both studies, we used our plugin to collect all source code classes and methods that participants selected or edited with in their IDE together with a timestamp for each interaction.

In the lab study, we recorded all reported activity descriptions, the time participants switched to each of these activities, the code elements they interacted with for each activity, and the code elements they found relevant for each activity.

For the field study, we recorded the activity descriptions that participants reported, the starting point, i.e. the first interaction that participants identified and the code elements they selected as being relevant for each activity. In total, we gathered data on 37 activities from the nine professional developers. For one developer we captured 5 activities since he voluntarily continued for a further activity. The professional developers in the field study worked on average 13.06 ($\pm$9.37) minutes on an activity before they got interrupted by the experimenter. The student developers, who reported the activities as they were investigating the change tasks, started a new activity on average each 4.84 ($\pm$3.77) minutes.

IV. Activity Characteristics (RQ1)

A first step towards activity-aware tool support is to better understand the characteristics and types of activities that developers break their work on a change task into.

Data Analysis. To investigate the characteristics and types of activities, we analyzed the self-reports from our participants. Overall, we collected a total of 96 activities: 37 from the field and 59 from the lab study. For these activities, the authors of this paper first applied an open coding approach to group the reported activities into distinct types of activities. A cross-validation of the activity types by another researcher not involved in this project resulted in an agreement of 92.7% of the activities with the remaining 7.3% being spread across categories. In a second step, we analyzed the number and relevance of code elements that participants interacted with for each self-reported activity. In the following, we report the averages across developers accompanied by the standard deviation denoted as '$\pm$'.

Activity Types. Based on the open coding of the 96 collected activities, we identified six distinct activity types, ranging from the search for a specific string to the changing of test case code. Table II presents details on each of these six activity types. For 4 of the 96 collected activity descriptions, we were not able to categorize them due to their vague descriptions.

The first two types we identified address *changes to source code* and differ in the code that was being changed, as the participants explicitly mentioned if they were working on test code. For *changing functionality*, activity descriptions ranged from extending to creating and adding functionality to the code, while *changes to test code* explicitly referred to test code being changed and the functionality that was being tested.

Two activity types refer to the *understanding of source code* and differ in the scope of investigation. While activity instances categorized as *understanding a specific code element* refer to developers trying to understand specific classes or methods, instances of *understanding a larger context* refer to whole features that spread across multiple code elements or large parts of a system that were investigated to locate the root of an undesired behavior or to understand the cause of a change. While the main focus of these two types is on the understanding of code, developers did not just navigate, search, and debug the code, but in several instances also edited the code during their work on these units.

The remaining two types only occurred in the lab study. One type refers to the *examination of a change task* that captured the reading/understanding of the task description and in forming the initial strategy to tackle the change task. The other type focuses on the *search for a specific String*, in particular, the use of a text/code search tool to locate a specific place in the source code. Several field study participants also searched the source code during their work, but different to lab study participants, they did not explicitly differentiate these searches as separate activities. This difference might be due to developers' familiarity with the source code in the field study and the shorter time spend on searching the code.

Overall, our findings provide evidence on a set of reoccurring activity types into which multiple developers decomposed change tasks into, whether in the lab on open source tasks or in the field on their usual tasks. The activity types we identified in our studies also overlap with several developer activities identified in previous research as illustrated in Figure 1 (bold ones are the ones that overlap), providing further evidence for the generalizability of the identified activity types. At the same time, several activities identified in earlier studies were not captured by our identified activity types, which stems from two reasons. First, the level of the activities that participants chose in their self-reports is at a higher level than some of the lower-

TABLE II: The six activity types identified in our two studies together with the number of reported instances of each activity type (# instances), the number of different developers that reported them (#devs), and exemplary instances.

Activity type	#instances field | lab	#devs field | lab	Exemplary instances
Changes to source code			
#1 Change functionality	8 | 5	6 | 4	(P11): implementation of the permission value connection (S7): Add config flag for global prefix
#2 Change test case code	6 | 10	4 | 7	(P8): Test the upload of user data (S6): Write new test to test the erroneous behavior [..]
Understanding source code			
#3 Underst. a specific code element	13 | 17	6 | 9	(P2): Check how to read the property pcy.[..].writer (S4): Try to understand Gson.create() method
#4 Underst. a larger context	6 | 17	4 | 9	(P9): [..] why is the data not read correctly? (S4): Find out how I can generate the output that is given [..]
#5 Change task examination	0 | 4	0 | 4	(S10): Inspect task
#6 Searching for specific string	0 | 6	0 | 4	(S11): Search for the setPrettyPrinting
Uncategorized	4 | 0	3 | 0	
	37 | 59		

level developer activities identified in previous studies. For instance, while participants navigated the source code using a variety of tools, shortcuts and views, none of them self-reported an activity which focused solely on the navigation activity, but rather used navigation steps in their activities. Second, due to the exploratory nature and the limited number of change tasks captured in our studies, we are not covering the complete range of activities that developers might decompose change tasks into. For instance, none of our participants chose to read documentation or code comments as identified by previous researchers [2], [12], [14]. Further studies are needed to extend our set of activity types and characteristics.

Activity Size. The size of self-reported activities is consistently small, with 3.5 ($\pm$3.0) unique classes and 5.2 ($\pm$4.1) methods that a developer interacted with per activity. This small number of selected and edited source code elements is consistent across all participants and reported activities, with only minimal differences between the professional developers working on their usual tasks in the field study (3.5 $\pm$3.4 classes, 5.0 $\pm$3.6 methods) and the students working on given change tasks (3.5 $\pm$2.8 classes, 5.4 $\pm$4.4 methods). For a fairer comparison, the calculation of these average values does not include activities of the type *change task examination* and *searching for a specific String*, as activities of these two types either included none or very few interactions within the code editor[1]. The activity size also did not vary significantly with the participants' programming experience (Pearson's $r = 0.3$). Only the time participants took to work on an activity in the lab study had a moderate effect on the activity size that was statistically significant.

Overall, these results suggest that developers decompose change tasks into relatively similar-sized and small activities, regardless of the vast differences in the change tasks they were working on and the number of explored elements. In contrast, the number of code elements developers explore for a change task can vary considerably from just a few to over hundred [5] and the number of methods changed can also vary considerably, ranging from a few to nearly 50 [28].

Relevant Code Elements. Across both studies and all reported activities, participants identified only 38.4% of the explored code elements—methods and classes—as relevant to the activity. Developers in the field navigated on average to 27.9 ($\pm$23.0) code elements (including revisits) and to 8.5 unique code elements out of which they determined 2.3 ($\pm$2.1) code elements to be relevant. Developers in the lab visited, including revisits, 18.0 ($\pm$14.5) code elements and 8.9 unique code elements. They determined 2.4 ($\pm$1.3) of these code elements to be relevant. This number is also independent of the time spent on the activity (no significant Pearson correlation) and independent from the number of code elements visited (no significant Pearson correlation) in both studies. Participants mentioned during the study that the irrelevant code elements were often captured when debugging, scrolling files, or when following traces that turned out to be unimportant.

Professional developers discovered the code elements which they found relevant on average after 9.79 ($\pm$10.07) navigation steps and students on average after 4.73 ($\pm$6.86) navigation steps. There is no correlation between developers' experience and the navigation steps performed until relevant code elements were found, meaning that in our analysis, the experience did not account for finding the relevant places in the source code faster. While professional developers found significantly more methods than classes relevant for an activity ($t(36) = 2.9, p = .006$), student developers found about an equal amount of methods and classes relevant.

[1]Including the activities of the type *searching a specific String* results in 3.3 ($\pm$2.7) unique classes and 5.0 ($\pm$4.4) unique methods.

Developers split change tasks into small and similarly-sized activities that can be categorized into a small set of recurring activity types. For each activity, a developer explored an average of 8.7 code elements of which a third is relevant.

V. Detecting Activity Boundaries and Type (RQ2)

To provide activity-aware tool support, we need to be able to detect when a developer switches between activities for a change task (RQ2a) as well as which type of activity the developer is working on (RQ2b). To investigate these two research questions, we analyzed the characteristics of developers' code interaction behavior. In particular, we examined characteristics related to the frequency and recency of code interactions, the relations between successively visited code elements and the edit history (see Table III). Several of these variables have previously been used in other studies to characterize developers' navigation behavior, as indicated in the table.

A. Boundary Detection

Data Analysis. To detect when a developer started to work on another activity (i.e. an activity switch), we looked at each source code method a developer interacted with and performed forward stepwise logistic regression, as we did not know which variables were reliable predictors [40]. We used the fact whether or not the method interaction was the start of a new activity as dependent variable and the characteristics of the method and previously visited methods as independent variables. For the variables str_step, lex_step, $field\_step$, $sameClass\_step$ (Var 2 to 5 in Table III), we examined up to four interactions back in time[2]. For each variable we counted the positive occurrences. We further calculated the variables t, rec, $freq$ (Var 8-10 in Table III), resulting in 19 ((4 x 4) + 3) independent variables. We filtered very short activities—activities that comprised less than four interactions with source code methods—resulting in 4 of the 37 reported activities from the field and 15 of the 59 reported activities from the lab study being excluded. In particular, all instances of the activity types *searching a specific String* and *change task examination* were excluded due to their shortness. In total, we analyzed 420 method interactions that included 33 reported activity switches from the field study and 435 method interactions from the lab study that included 44 reported activity switches.

Results. The results of our regression models show that developers' code interactions change when they switch to another activity and that we can use this to automatically detect activity switches based on logged interaction history with high accuracy. For the *field*, the final model of the stepwise logistic regression that we applied recognized 96.0% of the method interactions correctly as an activity switch or not. Furthermore, the model correctly detected 25 of the 33 (75.8%) activity switches. The variables that contributed significantly to the detection of the activity switches were the time t elapsed since the last interaction ($b = 0.008, p < .01$), and whether developers suddenly selected methods that were not using the same fields anymore $field\_step$ over the last four navigation steps ($b = -1.7, p = .033$). The final model significantly improved upon the baseline model ($\chi^2(2) = 123.288, p < .001$).

[2]In the exploratory analysis, we also looked further back in the interaction history, but since going back further than 4 interactions did not change the prediction accuracy significantly, we limited it to 4 steps back.

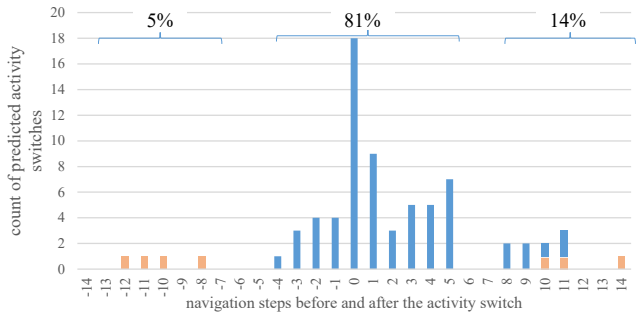


Fig. 3: Activity switch detection (lab): Distance of predicted to recorded activity switch position, which is at position '0'.

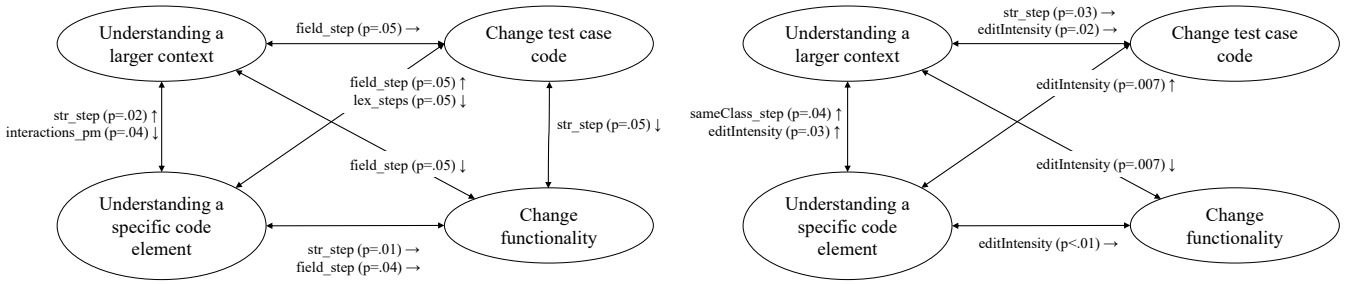
For the *lab*, the stepwise logistic regression resulted in a model that is able to correctly predict whether a navigation step is a switch or not in 81.4% of the cases. Out of the 44 explicitly denoted switches, the model recognized 18 (40.9%). The variables that contribute significantly to the final model were whether there is a call relationship to the previously explored method str_step ($b = 0.89, p = .05$), and whether the developer remained in the same class over the last three navigation steps $sameClass\_step$ ($b = 0.97, p = .02$). Our final prediction model again effectuated a significant decrease in unexplained variance compared to the base model, which only includes the intercept ($\chi^2(2) = 9.650, p = .008$). To further investigate the distance between predicted and recorded switches, we depicted the frequencies of the distances in Figure 3. Our model predicted 73 method interactions as switches, with 81% of the predicted switches being very close (less than 5 interactions away) from the one reported by the participants. Part of this slight discrepancy between predicted and recorded switch might be explained by a switch not being at the exact point when a developer reported it, which is also difficult to detect manually. Also, most of the predicted switches that are far away from the recorded activity switch stem from a single participant. These are marked in orange in Figure 3 and further investigation is needed to examine this.

B. Type Detection

Data Analysis. To analyze the feasibility and accuracy of automatically detecting the types of activities as well as which characteristics are most predictive, we performed a multinomial logistic regression over four of the six activity types. We excluded the types *change task examination* and the *search for a specific String*, since both of these types can be detected without analyzing the code navigation behavior. During *change task examination*, developers had no interactions with the code and were only looking at the task description, which can be detected automatically. During *String searches*,

TABLE III: Variables used to describe developers' navigation behavior.

Variable	VarID	Description
$interactions\_pm$	1	The amount of interactions within the source code per minute.
str_step	2	Determines whether a call relationship is existent between two methods, e.g., [41]
lex_step	3	Determines the cosine similarity between two methods, e.g., [5]
$field\_step$	4	Determines whether two methods use the same field, e.g., [41]
$sameClass\_step$	5	Determines whether two methods are defined within the same class, e.g., [5]
$isEdited$	6	Determines whether the method was edited, e.g., [9]
$editIntensity$	7	Determines the amount of characters which were changed within the method, e.g., [9]
t	8	The time elapsed since the last interaction, e.g., [26]
rec	9	Determines how recent (in terms of navigation steps) a method was selected, e.g., [9]
$freq$	10	Determines how frequent a method was selected, e.g., [9]



(a) Field study.

(b) Lab study.

Fig. 4: Significant variables for predicting an activity type in the field and lab study. The arrows next to the variable names point to the types in which the variable value is higher.

developers had much less code interactions (on average only 1.7 $\pm$1.2 code methods) than during their work on the other four activity types and interacted with a search tool in the IDE, which can also be detected automatically.

For the regression analysis, we used the characteristics and relations between successively visited methods and the edit history (Var 1-8 in Table III) as independent variables to predict the activity type (dependent variable). We calculated these characteristics for the method navigations a developer performed for an activity since she started working on the activity. In particular, we calculated the relative frequencies of the different kinds of navigation relations during the work on an activity, the average cosine similarity between subsequently selected methods, if a method was edited or not, and the amount of characters changed overall.

Results. The results of our analysis show that we can automatically detect different activity types with high accuracy by again using characteristics of developers' code interaction history. For the *field*, the final model of the performed regression analysis achieved a total prediction accuracy of 82%, with only few false predictions of each activity type. Compared to a baseline model, i.e. a model that omits all variables and only uses the intercept, the final regression model achieved a significant decrease in unexplained variance of $\chi^2(12) = 56.21, p < .001$. The step-wise multinomial logistic regression analysis showed that each of the four variables str_step ($p < .001$), $field\_step$ ($p < .001$), $interactions\_pm$ ($p < .001$), and lex_step ($p = .023$) provides a significant contribution to the final model for predicting the activity type. Figure 4a presents the variables that provide a significant contribution to distinguish between pairs of activity types, with the arrows indicating for which type a variable is higher. For example, developers who were working on *understanding a specific code element* interacted with more elements per minute than when they were *understanding a larger context* in the source code, but followed generally less call dependencies (str_step).

For the *lab*, the multinomial logistic regression that we carried out on the 49 gathered activities resulted in a model with 79.6% accuracy with only few activity types being predicted incorrectly. The final regression model was better than the baseline model and achieved a significant decrease in unexplained variance of $\chi^2(12) = 68.03, p < .001$. The distinguishing variables in this model were str_step ($p = .022$), $field\_step$ ($p = .049$), $sameClass\_step$ ($p = .044$), and $editIntensity$ ($p < .001$). Figure 4b shows the variables that help significantly to classify between two types of activities.

While these models provide a good indication of the feasibility and accuracy that can be achieved, especially since the accuracy is similar in both studies, and the variables indicate some of the differences in activity types, more data is needed to make these results more generalizable. For instance, the amount of changed characters ($editIntensity$) was a significant variable in the lab but not in the field study. We believe that lab study participants' unfamiliarity with the system and the substantially shorter time they spent on an activity, might have led to faster code changes and trial and error behavior in the lab that could account for this difference. Another reason for the difference in the variables used for

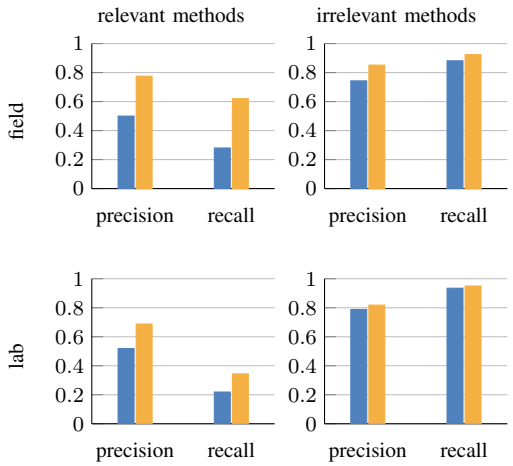


Fig. 5: Identification of relevant and irrelevant methods without (■) and with (■) activity information.

the final model are the uneven distribution across the activity types in the studies. Since only 10.2% of the activities were of type *change functionality* in the lab study and the other 3 types each occurred in more than 20% of the cases, the significance of the variables might have been influenced.

> Activity boundaries and activity type can be detected automatically and with high accuracy using fine-grained characteristics of developers' code navigation.

VI. Activity-Aware Relevancy Models (RQ3)

To explore the value of activity information for providing better tool support to developers, we focus on the identification of relevant code elements—a scenario commonly addressed by researchers [9]–[11], [42], [43]—and examine whether activity information improves the precision and recall of automatically identifying relevant code elements.

Data Analysis. To compare the detection of relevant elements with and without knowledge on the activities, we performed logistic regression and trained two types of classifiers: a base case and a set of activity-aware classifiers. For the base case, we used participants' relevance assessments of code elements as the dependent variable and characteristics of developers' code interactions as independent variables. For the independent variables, we focused on characteristics that previous research suggested for predicting relevant code elements. In particular, we calculated the variables str_step, $field\_step$, $sameClass\_step$, lex_step, $isEdited$, $editIntensity$, rec, and $freq$ (Var 2-7, 9 and 10 of Table III) for each method a participant interacted with.

To simulate activity-aware relevance prediction, we used the same dependent and independent variables as for the base case, but instead of just training one classifier over all code interactions in the training set, we trained one classifier per activity type and only over the the code interactions that were performed during the specific activity type in the training set. For these activity-aware classifiers, we focused on the four activities *change functionality*, *change test case code*, *understanding a specific code element*, and *understanding a larger context*, since activities of the other two types were again too short (having too few code interactions).

Overall, we performed a 10-fold cross validation using a random 90% of the data for training and the remaining 10% for testing. To compare the prediction of relevant code elements, we then predicted the relevance for each source code method in the test set once with the base case classifier and once with the classifier of the respective activity type. By having one classifier per activity type, we simulate an activity-aware classifier that knows which type of activity a developer is currently performing and when she is switching.

Results. The results of our analysis show that activity information considerably improved precision and recall for identifying relevant and irrelevant methods within developers' navigation histories (see Figure 5). For the *field study*, the precision of identifying relevant source code methods increased by 55.0% and the recall by 121.43%. For the *lab study*, the precision and recall to identify relevant methods increased by 32.59% and 57.14% respectively. The results of our analysis of relevant methods also shows that different variables play a more or less significant role in different activity types. For example, developers are more interested in structurally connected methods when they are *changing functionality* but less so when they are trying to *understand a specific code element*. While our analysis was conducted retrospectively, the insights into the contribution of different variables to the regression models for different activity types can also be used in the future to improve online recommendations on where to navigate next, especially in combination with the automatic detection of activity types.

> Knowledge of developers' activities can be used to improve tool support for change tasks.

VII. Threats to Validity

External Validity. There are several threats to the external validity of our results, in particular the limited number of study participants and change tasks in the lab study, the focus on one programming language and IDE and the unfamiliarity of the lab participants with the project. We tried to mitigate these threats by combining the controlled lab study with a field study in which professional developers worked on their own change tasks and their usual projects that differed in size and domain, and by choosing realistic change tasks for the lab study and a study system which is comparatively well commented.

Internal Validity. The presence of the experimenter, the regular interruptions, the writing of activity descriptions and the discomfort of being observed during the study might have influenced the participants' navigation behavior. By restricting the interaction with the participants during the study to a minimum, we tried to minimize the effect of this threat.

Construct Validity. The open coding of the gathered activity descriptions might contain a subjective bias. To mitigate this risk, all authors of this paper independently determined activity types after iterating through the activity descriptions. One of the authors then coded the activity into the emerged categories. To cross-validate our coding, we asked a working colleague to do the same. The two codings overlapped in most cases and in the few cases it did not, we discussed and finally agreed on one that was then used for our analysis.

VIII. Discussion

Our findings confirm and extend the existing body of research on how developers decompose tasks into the smaller activities they think and work in. The findings also demonstrate the potential of automatically detecting these activities and on the beneficial use of activity information. In the following, we will discuss the runtime detection of activities and how we plan to use activities to support developers in their work.

A. Runtime Detection

The earlier we can determine the type and boundaries of activities, the better we can take advantage of the information and support developers in their work. Our findings already demonstrate that it is possible to automatically detect activity boundaries with high accuracy at runtime by only taking into account developers' past code interactions. For the activity type detection on the other hand, we performed an analysis that uses all code interactions for an activity and could thus only be done retrospectively after the activity is completed. In a preliminary analysis that only considered the first five code interactions of the approximately 28 that a developer has per activity, we found that it is possible to predict the activity type with an accuracy of more than 63% for both, the field and the lab study. While further analysis is needed, these results already suggest that we are able to achieve a high accuracy for the automatic detection and take advantage of it early on.

B. Better Developer Support

Dynamic Adaptation of Artifact Recommendations. Several approaches have been proposed to support developers during change tasks by recommending various kinds of artifacts, ranging from specific source code elements, all the way to posts on Q&A sites, such as Stackoverflow. While all of these recommendations can be useful at some point during a change task, providing too many of them can limit the usefulness and lead to information overload. Taking into account when a developer starts to work on a particular activity type, we should be able to better tailor recommendations to the developers' information needs at any point in time and improve the usefulness of the recommendations. For example, while documentation might be more valuable when a developer is trying to understand a larger context, specific code snippets might be more useful when she is changing functionality. Similarly, activity information could be used to dynamically tailor views and the presented code context within an IDE that is often not adequate to support software developers as Minelli et al. [44] pointed out. Our analysis of activity types showed that different kinds of code information are relevant for different activity types. For instance, when a developer is changing functionality, views in the IDE could be organized to highlight the structural relations that are particularly relevant for this activity type.

Interruptions and Task Resumption. Interruptions by coworkers, instant messages or email occur frequently for software developers. When these interruptions happen at an inappropriate time, it takes the developer a comparatively high effort to get back into the train of thought and errors are more likely to happen [45]. Previous research on interruptions has already shown that interruption costs are considerably lower when developers switch between (sub-)tasks than when they are in the middle of a task, rendering these switches as more appropriate moments for interruptions [46]. Since entire change tasks can last several hours or even days, deferring an interruption until the next task switch might not be feasible. Activity switches are a lot more frequent (approximately every 8.4 minutes in our studies), denote switches that developers perceive in their work, and can be detected automatically with high accuracy. This suggests that activity switches represent good moments for interruptions and might be used to minimize required effort and potential errors during work.

At the same time, interruptions at inopportune moments are not completely avoidable. Information on activity boundaries, types and the relevant elements within could be used to ease task resumption. For instance, the activity information could be used to provide a better overview of the work to be resumed, by highlighting the relevant elements, or by presenting a more high-level activity view of the code interaction.

IX. Conclusion

In a first step towards activity-aware tool support for change tasks, we investigated the activities that developers break their work on a change task into. We conducted two exploratory studies, a lab and a field study with a total of 21 software developers and examined the characteristics, the automatic detection and the potential value of these activities and the knowledge thereof. Our results show that activities are consistently small across developers and change tasks and that few of the code elements that developers interacted with during an activity are relevant. Our analysis also showed that fine-grained navigation behavior of developers can be used to accurately detect activity boundaries and types.

The newly gained insights on the activities of developers and their automatic detection represent valuable opportunities to better support developers in their work. In particular, this information can be used to improve the detection and recommendation of relevant code elements and artifacts as well as for better interruption management and task resumption. A case study we performed on the detection of relevant code elements has shown that this information can be used to improve upon more traditional approaches by 33% for precision and 57% for recall, indicating the potential that the detection of activities can have on developer support.

REFERENCES

[1] D. E. Perry, N. Staudenmayer, and L. G. Votta, "People, Organizations, and Process Improvement," *IEEE Software*, vol. 11, no. 4, pp. 36–45, Jul. 1994.

[2] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[3] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, 2006, pp. 492–501.

[4] J. Sillito, G. C. Murphy, and K. De Volder, "Questions Programmers Ask During Software Evolution Tasks," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '06, 2006, pp. 23–34.

[5] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' Code Context Models for Change Tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14, 2014, pp. 7–18.

[6] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 344–353.

[7] R. Minelli, A. M. and, and M. Lanza, "I Know What You Did Last Summer: An Investigation of How Developers Spend Their Time," in *Proceedings of the 23rd International Conference on Program Comprehension*, ser. ICPC '15, 2015, pp. 25–35.

[8] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A Study of Visual Studio Usage in Practice," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, 2016, pp. 124–134.

[9] M. Kersten and G. C. Murphy, "Using Task Context to Improve Programmer Productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '06/FSE '06, 2006, pp. 1–11.

[10] M. J. Coblenz, A. J. Ko, and B. A. Myers, "JASPER: An Eclipse Plug-in to Facilitate Software Maintenance Tasks," in *OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '06, 2006, pp. 65–69.

[11] M. P. Robillard and F. Weigand-Warr, "ConcernMapper: Simple View-based Separation of Scattered Concerns," in *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*, ser. eclipse '05, 2005, pp. 65–69.

[12] A. M. Vans, A. von Mayrhauser, and G. Somlo, "Program understanding behavior during corrective maintenance of large-scale software," *International Journal of Human-Computer Studies*, vol. 51, no. 1, pp. 31 – 70, 1999.

[13] A. J. Ko and B. A. Myers, "A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems," *Journal of Visual Languages and Computing*, vol. 16, no. 1-2, pp. 41–84, Feb. 2005.

[14] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '97, 1997, pp. 21–36.

[15] R. Brooks, "Towards a Theory of the Cognitive Processes in Computer Programming," *International Journal of Human-Computer Studies*, vol. 51, no. 2, pp. 197 – 211, 1999.

[16] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. van der Hoek, "Microtask Programming: Building Software with a Crowd," in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '14, 2014, pp. 43–54.

[17] V. Rajlich and P. Gosavi, "Incremental Change in Object-Oriented Programming," *IEEE Software*, vol. 21, no. 4, pp. 62–69, Jul. 2004.

[18] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, Jul. 2006.

[19] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?" in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 79–103.

[20] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig, "A Comparative Study of Manual and Automated Refactorings," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, ser. ECOOP'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 552–576.

[21] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, How, and Why Developers (Do Not) Test in Their IDEs," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 179–190.

[22] K. Kevic, B. M. Walters, T. R. Shaffer, B. Sharif, D. C. Shepherd, and T. Fritz, "Tracing Software Developers' Eyes and Interactions for Change Tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '15, 2015, pp. 202–213.

[23] Z. Soh, A. Yamashita, F. Khomh, and Y. G. Guhneuc, "Do Code Smells Impact the Effort of Different Maintenance Programming Activities?" in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, ser. SANER '16, vol. 1, March 2016, pp. 393–402.

[24] I. Safer and G. C. Murphy, "Comparing Episodic and Semantic Interfaces for Task Boundary Identification," in *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, ser. CASCON '07. Riverton, NJ, USA: IBM Corp., 2007, pp. 229–243.

[25] S. Stumpf, X. Bao, A. Dragunov, T. G. Dietterich, J. Herlocker, L. Li, and J. Shen, "Predicting User Tasks: I Know What Youre Doing," in *In 20th National Conference on Artificial Intelligence, Workshop on Human Comprehensible Machine Learning*, ser. AAAI '05. Press, 2005.

[26] I. D. Coman and A. Sillitti, "Automated Identification of Tasks in Development Sessions," in *16th IEEE International Conference on Program Comprehension*, ser. ICPC '08. IEEE Computer Society, 2008, pp. 212–217.

[27] L. Zou and M. W. Godfrey, "An industrial case study of Coman's automated task detection algorithm: What Worked, What Didn't, and Why," in *Proceedings of the 28th International Conference on Software Maintenance*, ser. ICSM '12, Sept 2012, pp. 6–14.

[28] M. Barnett, C. Bird, J. a. Brunet, and S. K. Lahiri, "Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 134–144.

[29] M. P. Robillard, "Automatic Generation of Suggestions for Program Investigation," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 11–20, Sep. 2005.

[30] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program Understanding and the Concept Assignment Problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, May 1994.

[31] T. D. LaToza and B. A. Myers, "Visualizing Call Graphs," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '11, Sept 2011, pp. 117–124.

[32] V. Augustine, P. Francis, X. Qu, D. Shepherd, W. Snipes, C. Bräunlich, and T. Fritz, "A Field Study on Fostering Structural Navigation with Prodet," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 229–238.

[33] J. Lawrance, R. Bellamy, and M. Burnett, "Scents in Programs: Does Information Foraging Theory Apply to Program Maintenance?" in *IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '07, Sept 2007, pp. 15–22.

[34] C. Parnin and C. Gorg, "Building Usage Contexts During Program Comprehension," in *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ser. ICPC '06, 2006, pp. 13–22.

[35] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting Source Code Changes by Mining Change History," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, Sept 2004.

[36] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining Version Histories to Guide Software Changes," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04, 2004, pp. 563–572.

[37] E. Hill, L. Pollock, and K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, 2007, pp. 14–23.

[38] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, "Modeling Programmer Navigation: A Head-to-Head Empirical Evaluation of Predictive Models," in *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, ser. VL/HCC '11, Sept 2011, pp. 109–116.

[39] Gson, "A Java Serialization/Deserialization Library to Convert Java Objects into JSON and Back," https://github.com/google/gson, 2016, accessed online: 2017-01-06.

[40] A. Field, *Discovering Statistics Using SPSS*. SAGE Publications, 2005.

[41] M. P. Robillard and G. C. Murphy, "Automatically inferring concern code from program investigation activities," in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ser. ASE '03, Oct 2003, pp. 225–234.

[42] B. Biegel, S. Baltes, I. Scarpellini, and S. Diehl, "CodeBasket: Making Developers' Mental Model Visible and Explorable," in *Proceedings of the 2nd International Workshop on Context for Software Development*, ser. CSD '15, 2015, pp. 20–24.

[43] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '10, 2010, pp. 2503–2512.

[44] R. Minelli, A. Mocci, R. Robbes, and M. Lanza, "Taming the IDE with Fine-Grained Interaction Data," in *2016 IEEE 24th International Conference on Program Comprehension*, ser. ICPC '16, May 2016, pp. 1–10.

[45] B. P. Bailey and J. A. Konstan, "On the Need for Attention-Aware Systems: Measuring Effects of Interruption on Task Performance, Error Rate, and Affective State," *Computers in Human Behavior*, vol. 22, no. 4, pp. 685 – 708, 2006, Special issue: Attention Aware Systems.

[46] S. T. Iqbal, P. D. Adamczyk, X. S. Zheng, and B. P. Bailey, "Changes in Mental Workload During Task Execution," in *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '04, 2004.